Universidade de Aveiro

Projeto 2

João Fernandes (93460) João Martins (93183) Miguel Ferreira (93419)

Projeto em Informação e Codificação

Departamento de Eletrónica, Telecomunicações e Informática

29 de dezembro de 2021

Conteúdo

Lista de Figuras

1 Introdução

Relatório do segundo projeto de Informação e Codificação onde iremos codificar imagens e som em código Golomb. Iremos usar **AudioFile.h** para manipular ficheiros de áudio e **OpenCV** para as imagens.
Na **Parte A** criamos a classe BitStream que nos permite ler e escrever bit a bit e a classe Golomb que irá servir para codificar e descodificar.
Na **Parte B** e **Parte C** usámos o algoritmo Golomb e o Bitstream criados na **Parte A** para codificar e descodificar *samples* de audio e imagens respetivamente.
Git Repository

2 Parte A - BitStream and Golomb

2.1 BitStream

Uma classe foi implementada com os seguintes argumentos:

1. **fname** - nome do ficheiro para ler ou escrever
2. **m** - modo em que o utilizador inicializa a classe. **w** para escrever e **r** para ler

A partir daqui, dependendo no modo escolhido, pode-se usar os seguintes métodos:

1. **writeBit** - Disponível apenas em modo escrita. Enche progressivamente um *bitBuffer* de 1 *byte* e escreve-o para o ficheiro quando encher. *Buffers* não cheios serão escritos ao fechar a *BitStream*.
2. **writeBits** - Disponível apenas em modo escrita. Usa o método *writeBit* para escrever para o ficheiro.
3. **readBit** - Disponível apenas em modo escrita. Começa por ler um *byte*, depois extrai o bit atual.
4. **readBits** - Disponível apenas em modo escrita. Usa o método *readBit* para ler a quantidade de bits (len) desejada.

2.2 Golomb

Calculamos o Golomb da seguinte maneira:

1. O M é dado como argumento
2. Com o N obtemos o quociente e o resto
   (a) q = floor(N/M)
   (b) r = N % M
3. Gerar a palavra codificada
   (a) O código devolverá uma string binária sendo o quociente e o resto codificados de maneiras distintas
   (b) Para codificar o quociente escrevemos em codigo unário com o numero de bits a 0 igual ao valor do quociente.
   (c) O resto é codificado em codigo binário truncado
      i. Começamos por fazer $b = \lfloor \log_2(M) \rfloor$
      ii. Se o resto $< 2^{b+1} - M$, codifica-se o resto em binário com b bits
      iii. Se o resto $\geq 2^{b+1} - M$ codifica-se o valor de $r + 2^{b+1} - M$ em binário com $b + 1$ bits

3 Parte B - Audio Codec

O codec de audio é constituido por 2 classes, *AudioCodec* e *AudioPredictor*, que implementam, respetivamente, a API para codificação e descodificação de ficheiros *wave* e o sistema para *predictive coding*.

3.1 Predictor

O preditor foi implementado usando a formula apresentada nos slides de *Predicitive Coding*.

$$\begin{cases} x\hat{}(0) & \text{n = 0} \\ x\hat{}(1) & \text{n = xn1} \\ x\hat{}(2) & \text{n = 2xn1 xn2} \\ x\hat{}(3) & \text{n = 3xn1 3xn2 + xn3} \end{cases}$$

O preditor pode ser configurado aquando da sua construção com a ordem, correspondente ao numero de *samples* anteriores a considerar.

```
AudioPredictor(int order=1);
```

Na sua implementação foi usado um *Buffer Circular* da biblioteca *Boost*.

3.2 Encoding

```
int encode(std::string finPath, std::string foutPath, ENC_OPT opts);
int encode(std::string finPath, std::string foutPath);
```

O processo de encoding pode ser configurado usando os atributos na seguinte *struct*:

```
typedef struct enc_options {
        bool histogram = false;
        string histogram_out_file = "";
        int quantization_factor = 0;
        int predictor_order =1;
        int samples_per_block=10;
        int starter_golomb_m = 4;
    } ENC_OPT;
```

Os primeiros dados a serem codificados são meta-dados necessários na descodificação:

- Numero de Canais;
- *Samples* por bloco;
- Oredem do *Predictor*;
- Fator de Quantização.

Primeiro é feito o encoding das primeiras *samples* de cada canal. Caso a ordem dos preditores, um por canal, seja maior do que 1, precisando de mais *samples* para preencher os seus *buffers*, as restantes *samples* são codificadas como se fossem um bloco.
O *encoding* das restantes *samples* de um ficheiro *wave* é feito por blocos de tamanho $samples_per_block * nCanaisAudio$ (Figura 1):
Em cada bloco são calculados os *residuals* de cada *sample* usando a fórmula $residual = sample - predictor()$. Caso o fator de quantização seja diferente de 0, o *residual* é ainda dividido por $2^{quantizationFactor}$.

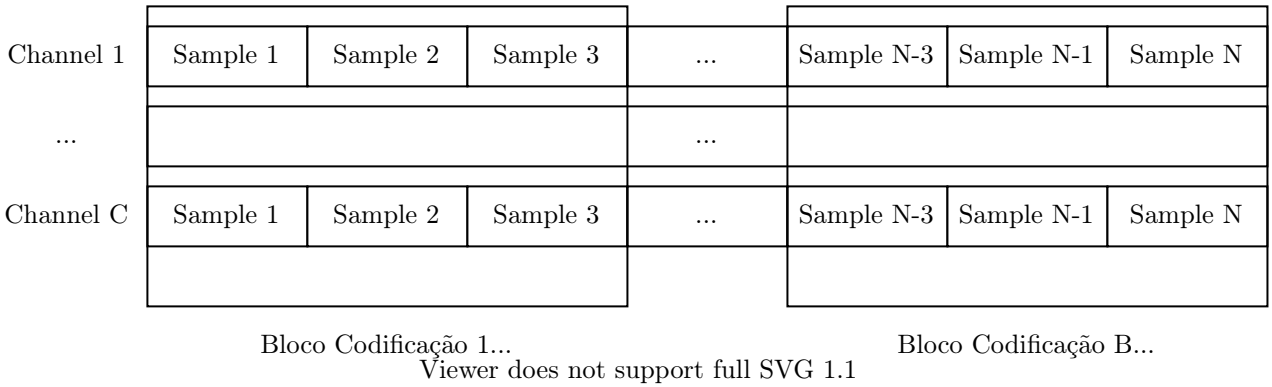

Figure 1: *Codificação por Blocos*

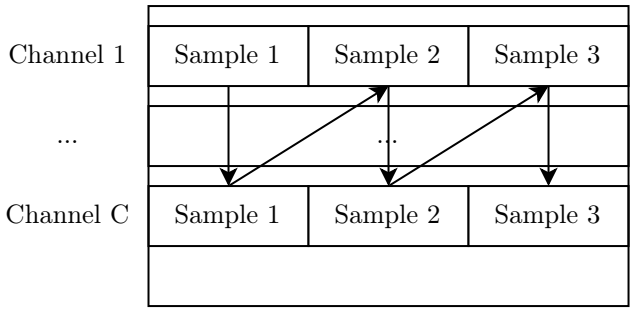

Figure 2: *Codificação de samples num bloco.*

A escolha das *sample* a serem analizadas a seguir é feita da seguinte forma de forma a simplificar a descodificação(Figura 2):
Após todas *samples* correspondentes a um bloco serem processadas, e calculada uma média dos residuals(valores absolutos) e com base nessa média é determinado um m ideal para os codificar usando codificação de *Golomb*. O novo m é codificado usando o m anterior, o codificador de *Golomb* é atualizado com o novo m e segue-se a codificação dos *residuals* do bloco.

3.3 Decoding

```
int decode(std::string finPath, std::string foutPath);
```

O *decoder* consiste, em suma, do processo inverso do *encoder*. Lê os valores descodificados e atribuilhes um significado de acordo com a ordem em que são lidos, portanto, começa por ler os metadados dando seguidamente o *set-up* do ambiente de descodificação.
Quanto à descodificação dos dados relativos às *samples* de audio começa por fazer a descodificação das primeiras *samples* e tambem das necessárias para fazer o correto *load* dos *buffers* dos predictors (um por canal) e depois começa a descodificar bloco a bloco até chegar ao fim do ficheiro de forma similar ao *encoder* (Figura 2).

3.4 Notas de Implementação

- Devido a necessidade de lidar tanto com samples em *float* como em *int*, de momento tanto o *encoder* como o *decoder* apenas suportam um *bit depth* de 16 bits, no entanto, é compativel com as *samples fornecidas*. A resolução consisteria na deteção do bitdepth*bitdepth* e a sua codifição usando numeros de 0 a 2 representativos dos diversos modos: 16 24 e 32 bit, respetivamente; e da sua correta conversão.
- A configuração do encoder não é controlavel diretamente pelo utilizador aquando da invocação do executavel, porque devido ao numero de argumentos seria necessario usar um *argparser*, no entanto é possivel modificar os valores da *struct* ma *main*.

3.5 Dependências

As seguintes dependencias estão incluidas na pasta */lib* do repositório, não necessario a sua instalação por parte do utilizador.

- AudioFile.h
- Biblioteca Boost (v1.78)

4 Parte C - Image Codec

Para fazer o encode

```
int encode(string img, string fileOut, string encodFormat,
           string predictor, int mode, int shifts);
```

Este **encoder** irá usar o **OpenCV** para percorrer a imagem, pixel a pixel, e recolher o valores RGB. Os valores RGB irão ser transformados em valores YUV que é o formato pedido no projeto. Cada pixel, já em YUV, irá passar por um dos predictors (*Linear Non-Linear*) que irá ajudar na codificação, pois é um valor "mais pequeno"mas muito próximo do original, ou seja, consome menos tempo a ser codificado, e na descodificação pois irá tornar mais fácil ler os valores e retorna-los como se encontravam antes da codificação. Porém o *predictor* não pode criar nenhum erro pois assim irá desformatar a imagem quando está for descodificada. Depois de obter o valor do *predictor* irá ser escolhida, pelo utilizador, a maneira de codificar(*Lossless* ou *Lossy*). Numa codificação *Lossless* o tempo de execução será bastante elevado mas toda a informação original irá estar presente. Numa codificação *Lossy* o tempo de execução será inferior mas, como diz no nome, alguma informação irá ser perdida durante a codificação o que fará com que a imagem fique com menor qualidade as ser descodificada.

4.1 RGB to YUV

Sendo que os valores obtidos originalmente estão no formato **RGB** fizemos a conversão para **YUV**.

```
void RGB_2_YUV(Vec3b RGBpixel){
    Y = 0.299*R + 0.587*G + 0.114*B;
    U = 0.492*(G - Y);
    V = 0.877*(R - Y);
}
```

4.2 Predictor

Para os predictors, usamos quatro pixeis como demonstrado na imagem abaixo.

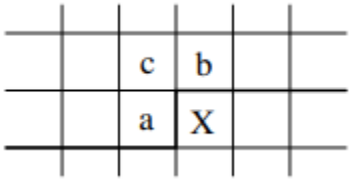

Figure 3: *Pixeis usados no predictor*

O X é o pixel que está a ser codificado.

O *Linear Predictor* implementado tem sete modos:

1. a
2. b
3. c
4. a + b - c
5. a + (b - c)/2
6. b + (a - c)/2
7. (a + b)/2

Para o *Nonlinear Predictor* usamos um algoritmo com base nestas condições.

- Se c $\geq$ max(a,b) então o resultado é obtido por min(a,b)
- Se c $\leq$ min(a,b) então o resultado é obtido por max(a,b)
- Caso contrário fica a + b - c

Sendo este um codificador **lossless**, passamos todos os bits pelo algoritmo de Golomb e imprimimos o resultado num ficheiro.

4.3 Melhor "M"para o Golomb

Para obtermos o melhor valor do M para o algoritmo de Golomb usamos a fórmula $M = \frac{-1}{\log_2((mean)/(mean+1.0))}$, sendo *mean* a média dos valores RGB do pixel.

4.4 Lossless Encoder

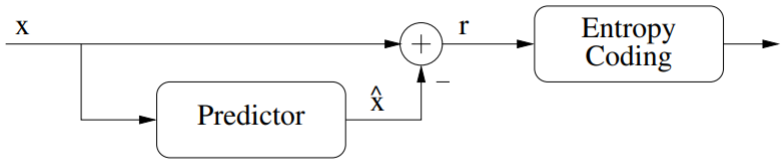

Figure 4: *Lossless encoder*

Para obtermos um encoder lossless comaçá-mos por calcular o valor ideal do M a ser coloca no *Golomb*. Com o valor obtido calculamos o código Golomb para o R, o G, e o B separadamente.

4.5 Lossy Encoder

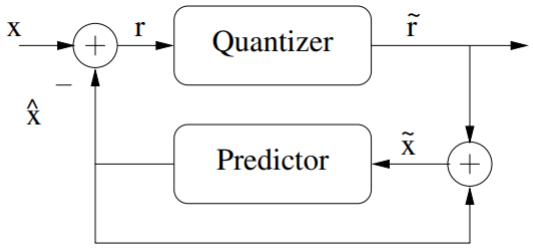

Figure 5: *Lossy encoder*

Para obter uma codificação lossy:

- começamos por calcular o *residual* que é a diferença entre o Pixel atual e o seu predictor;
- fazemos *shifts* para a direita no residual (a quantidade de shifts é recebida como arguemnto) ou seja, estamos a dividir o resultado do residual;
- criamos uma *new sample* que será igual à soma do residual ja com o shift e da prediction anterior.
- fazemos novamente a *prediction* mas com a *new sample*.
- finalmente passamos o valor da *prediction* final para o algoritmo de *Golomb* para ser codificado e inserido no ficheiro.

Para utilizar o encoder usamos a mesmo função acima demonstrada. É nesessario apenas especificar a entrada *encodFormat* como *lossy*.